PolicyDSL: A Model-Driven Approach for the Definition and Enforcement of Fine-Grained Access Control Policies

Atefeh Nirumand
Luxembourg Institute of Science and Technology
Esch-sur-Alzette, Luxembourg
atefeh.nirumand@list.lu

Jordi Cabot
Luxembourg Institute of Science and Technology
University of Luxembourg
Esch-sur-Alzette, Luxembourg
jordi.cabot@list.lu

Abstract

Access control represents a fundamental concern in software systems. Despite the extensive implementation of both Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC), their integration into Model-Driven Development (MDD) remains limited, particularly in supporting both multi-layer enforcement across back-end and front-end (UI) layers and comprising fine-grained control over interface elements. Given the growing importance of security in all types of applications, the need for such integration, and therefore, the model-driven development of secure by design applications has become paramount.

To overcome this situation, this paper proposes PolicyDSL, a Domain-Specific Language (DSL) and model-driven approach for multi-layer fine-grained access control. It defines a metamodel for data and GUI elements access control definitions together with run-time enforcement adaptations that guarantee the generated application satisfies the specified access control policies. The approach has been implemented on top of the BESSER low-code platform and evaluated on a use-case with 200 policy variants.

CCS Concepts

• **Security and privacy → Software security engineering**; **Web application security**; **Domain-specific security and privacy architectures**.

Keywords

Model-Driven Development (MDD), Low-Code Framework, Fine-Grained Access Control, Policy Modeling

ACM Reference Format:
Atefeh Nirumand and Jordi Cabot. 2026. PolicyDSL: A Model-Driven Approach for the Definition and Enforcement of Fine-Grained Access Control Policies. In *ACM/IEEE 29th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion 2026), October 04–09, 2026, Málaga, Spain.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3837062.3838897

1 Introduction

Access control represents a fundamental aspect of security in contemporary software systems, playing a crucial role in the protection of resources from unauthorized access [8, 21]. With the growth of data-intensive applications and the widespread adoption of distributed work environments, robust and flexible authorization mechanisms have become essential.

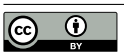
Research on access control models has shown a steady increase in publications in the past decade, reflecting its continued importance in software engineering. Among existing paradigms, including Discretionary Access Control (DAC) and Mandatory Access Control (MAC) to Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC), RBAC and ABAC remain the most widely utilized in enterprise systems [5, 9, 20].

In practice, access control in data-driven applications is often implemented at the code level using frameworks such as Spring Security or front-end authorization guards [9]. However, modern applications require consistent enforcement across both the back-end and the UI layers. Existing approaches [1, 4–6, 9, 14] tightly couple authorization logic with implementation details, increasing maintenance cost and reducing analyzability.

Moreover, existing model-driven approaches (e.g., SecureUML [13], IFML [17]) address either authorization or UI modeling, but not unified multi-layer policy specification and enforcement. Recent work [18] on conversational interfaces introduces partial RBAC support but does not generalize to graphical UI components. Fine-grained multi-layer access control remains under-explored in model-driven engineering.

Low-Code and No-Code (LCNC) platforms are a new style of model-driven development [7]. Despite its popularity (e.g. by 2027, around 75% of new applications are expected to be developed using low-code technologies[1]) [2], and impressive code-generation capabilities (mostly from a simplified modeling perspective focusing on domain and UI models for the generation of data-driven web applications) they also show the same limitations. Commercial low-code platforms (e.g., OutSystems, Mendix) provide access control features, but typically with limited and inconsistent granularity across UI and data layers [15].

This paper proposes a unified model-driven approach for specifying and enforcing access control across the core application layers within the BESSER [3] low-code framework. We introduce PolicyDSL, a platform-independent metamodel for defining fine-grained policies over both data and GUI elements in a single specification. Unlike existing approaches that focus on either back-end or coarse-grained UI control, PolicyDSL supports both multi-layer enforcement and fine-grained access control over application elements (e.g., screens, widgets, fields, and navigation). From this specification, the pipeline generates (i) back-end enforcement via



This work is licensed under a Creative Commons Attribution 4.0 International License.
MODELS Companion 2026, Málaga, Spain
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2903-4/2026/10
https://doi.org/10.1145/3837062.3838897

[1] https://www.gartner.com/en/documents/7146430

FastAPI middleware and (ii) front-end enforcement via a React-based policy context that dynamically adapts the UI. This approach ensures consistent authorization across layers and eliminates discrepancies between data and interface behaviour.

The approach produces complete data-driven applications from integrated models and supports incremental policy evolution. Policies can be authored either programmatically or through a Natural-Language (NL) interface that maps user requirements to the PolicyDSL model, reducing the barrier to policy specification. We further show that the extension of approach through AI-agent-aware access control, where agents are treated as first-class subjects and authorization combines user, agent, and delegation policies to enable controlled delegation. The approach is evaluated over 200 policy variants spanning coarse-, medium-, and fine-grained levels, covering correctness, robustness, and run-time performance. Results show correct multi-layer enforcement, robustness against adversarial inputs, and low run-time overhead across both layers.

The contributions of this paper are:

- PolicyDSL, a metamodel-based language with concrete syntax in Python and support for NL policy specification for fine-grained access control over UI and data resources.
- A model-driven pipeline that generates policy-aware applications with consistent enforcement across back-end and front-end layers.
- An open-source implementation of the full generation and enforcement workflow.
- An empirical evaluation over 200 policy variants assessing correctness, robustness, and run-time performance.

The remainder of the paper is organized as follows. Section 2 presents the motivation and running example. Section 3 describes the approach, followed by tool support in Section 4. Section 5 presents the evaluation, Section 6 reviews related work, and Section 7 introduces the AI-agent-aware extension. Section 8 discusses limitations and threats to validity, and Section 9 concludes the paper and outlines future work.

2 Motivation and Running Example

This section introduces the motivation for the present work and presents the running example used throughout the paper.

Role- and Attribute-Based Access Control. Access control mechanisms are designed to regulate the manner in which users interact with system resources. RBAC is a model that has been widely adopted. In this model, permissions are assigned to roles rather than individual users. This has the effect of simplifying administration and improving scalability [19]. ABAC extends this paradigm by enabling authorization decisions based on attributes of users, resources, and environmental conditions (e.g., time, location). While RBAC offers simplicity and structure, ABAC provides greater flexibility and supports fine-grained, context-aware policies. In practice, modern systems often adopt hybrid approaches that combine role-based and attribute-based constraints to achieve a balance between manageability and expressiveness [10].

Running Example. We consider a *Library Management* application with three roles: *Librarian*, *Faculty*, and *Student*. The system includes core entities such as *Book*, *Library*, and *Author*. As shown in Figure 1, a *Library* contains multiple *Book* entities, and each

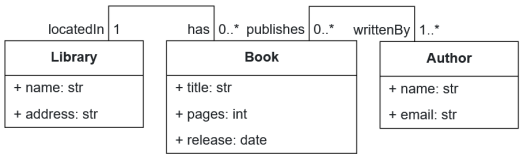


Figure 1: UML Class Diagram of the Library Management Structural Model.

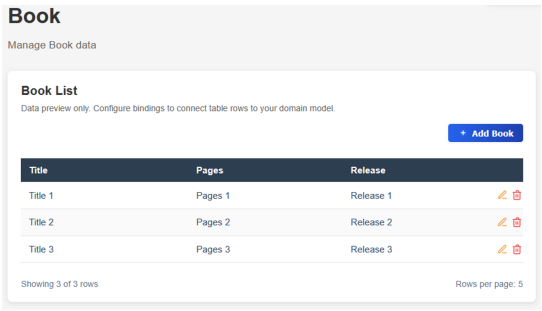


Figure 2: Book Page for the Library Management Application.

Book is associated with one or more *Author*s. For simplicity, each *Book* is associated with a single *Library*; however, the proposed approach (Section 3) is not restricted and supports arbitrary association multiplicities. The GUI model, rendered as the *Book* page in Figure 2, defines the main screens, including a *Book* screen with CRUD operations and data binding to the *Book* entity. Access control requirements are defined at multiple levels: a *Librarian* has full permissions; a *Faculty* member has restricted actions and masked sensitive fields (e.g., *Release date* and *Author email*); and a *Student* has further restricted or hidden screen access. These constraints propagate to the UI, where components are dynamically enabled, disabled, or masked based on role. This example illustrates fine-grained, cross-layer access control, which is typically implemented manually and is therefore error-prone due to scattered logic across back-end and front-end components. These limitations motivate a unified model-driven approach for consistent policy specification and enforcement across core application layers.

3 Our Approach

This section presents the two main components of our proposal, the PolicyDSL language (Section 3.1) and the generation architecture (Section 3.2) responsible for translating our policies into run-time consistent access control on top of both the data and GUI elements.

3.1 PolicyDSL Metamodel

This subsection presents the abstract syntax of PolicyDSL (Section 3.1.1) and the ways in which a user can author policies over it (Section 3.1.2).

3.1.1 Abstract Syntax. PolicyDSL (illustrated in Figure 3) is an extension of B-UML[2] (BESSER Universal Modeling Language) that

[2]https://besser.readthedocs.io/en/latest/buml_language.html

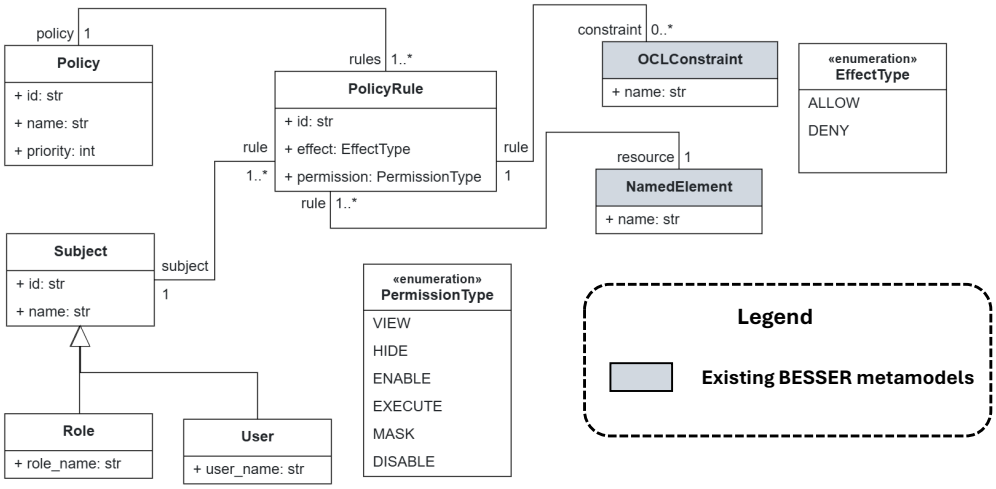


Figure 3: The PolicyDSL Metamodel.

incorporates a dedicated package for the specification of access control policies. The metamodel directly reuses existing B-UML metaclasses to ensure full traceability and consistency with the underlying domain and GUI models. In particular, PolicyDSL does not redefine structural or GUI concepts, but instead references them through the unified B-UML metaclass `NamedElement`, which serves as the root for all identifiable structural and GUI elements in the B-UML ecosystem. This metamodel was designed through an extensive review of existing approaches in the literature, including [4–6, 10, 12, 13, 15, 18]. The design is founded on three principles: (i) data- and GUI-level granularity, (ii) fine-grained permission types beyond CRUD, and (iii) separation of concerns between Policy, Structural, and GUI models. PolicyDSL facilitates the explicit referencing of Structural and GUI model elements (like screens, buttons, and input fields.). This enables the definition of policies at the same level at which interactions occur, thereby facilitating consistent enforcement across back-end and front-end layers.

The metamodel is centered around a concise set of fundamental concepts. A *PolicyRule* associates a subject, a target element (resource), a permission, optional constraints, and an effect. Since targets are B-UML `NamedElements`, PolicyDSL covers both data-level entities and GUI elements, including view containers (screens), UI components, actions, and data-bound elements such as fields or table columns. A *Subject* represents the actor within the system and is specialized into *User* and *Role*. The metamodel is designed to accommodate both RBAC and ABAC; however, the current implementation and evaluation focus is mainly on role-based policies. The *permission* attribute specifies the operation to be regulated and is typed by `PermissionType`. Besides standard CRUD operations, PolicyDSL supports UI-level permissions such as `VIEW`, `HIDE`, `ENABLE`, `DISABLE`, `MASK`, and `EXECUTE`, enabling fine-grained control of interface behaviour. Constraints are defined using the B-UML OCL metamodel. Each constraint is an `OCLConstraint` evaluated over elements of the underlying model, enabling contextual and data-dependent access control. OCL constraints currently apply at the rule level, evaluated over model elements. Extending them to the policy model (e.g., for cross-rule consistency) is a natural next step. Structural validity is ensured by the validator (Section 3.1.2); formal metamodel-level invariants are left for future work.

Table 1: Excerpt of PolicyDSL rules for the Library Management case study.

Subject	Resource	Permission	Effect	Level
Librarian	BookScreen	VIEW	ALLOW	Screen
Student	BookScreen	VIEW	DENY	Screen
Librarian	BookEditButton	ENABLE	ALLOW	Action
Faculty	BookEditButton	ENABLE	DENY	Action
Faculty	Book.release	MASK	ALLOW	Field
Faculty	Author.email	MASK	ALLOW	Field

A *Policy* groups multiple rules and is assigned a priority used to order evaluation when multiple policies apply. PolicyDSL adopts a deny-overrides strategy: if any rule evaluates to `DENY` for a given (Subject, Resource) pair, the final decision is `DENY`, regardless of other `ALLOW` decisions. Otherwise, access is `ALLOW` if at least one rule permits it, and not applicable otherwise. Policies are evaluated in descending priority order; however, priority only affects the evaluation sequence, while conflict resolution is determined by the deny-overrides policy. This separation ensures consistent enforcement across layers. PolicyDSL supports two enforcement modes: (i) back-end enforcement, where decisions are propagated to the UI, and (ii) front-end enforcement, where policies dynamically adapt the user interface.

3.1.2 Policy Authoring. Table 1 shows an excerpt of PolicyDSL rules for the *Library Management* example, covering screen-, action-, and field-level constraints. Rules can be authored in two ways.

Programmatic Authoring. As B-UML is the backbone of BESSER, policies can be defined in Python by instantiating metaclasses of the

PolicyDSL metamodel (e.g., *Role, PolicyRule, PermissionType, EffectType, Policy*) and referencing Structural and GUI `NamedElements`. This provides full control over the definition of the rules.

Natural-language Authoring. Alternatively, the user describes the intended access rules in natural language and an LLM performs the transformation into the same metaclass instances. To make this reliable, we use a carefully engineered prompt built with standard prompt-engineering practices: the prompt (Listing 1) extracts subjects, resources, permissions, and effects. For example, *"Students cannot view author email addresses"* is mapped to a *DENY* rule for the *Student* role with *VIEW* permission over *Author_email*. The generated rules are validated against the metamodel and directly integrated into the existing pipeline, reducing the barrier to policy definition while preserving correctness. A single NL request can also grant a permission to several elements at once (e.g., masking a set of fields), which the transformation expands into the corresponding set of individual rules through a *field_set* target. A preliminary evaluation (Section 5.4) indicates that the LLM correctly performs this transformation for the running example; a broader study is left for future work.

```
Task: Convert the natural-language access-control
description into a structured policy draft for BESSER.
Identify:
- roles (subjects)
- resources (e.g., entity, field, or screen)
- permissions (e.g., VIEW, HIDE, ENABLE, MASK)
- effect (ALLOW or DENY)
Return a list of policy rules, where each rule specifies:
(role, resource, permission, effect).
```

Listing 1: LLM prompt for NL policy authoring (simplified due to space limitations).

Regardless of the authoring path, the policy model is validated before generation. The validator detects (i) duplicate rule identifiers, (ii) conflicting rules on the same subject–resource pair (e.g., `VIEW/HIDE`, `ENABLE/DISABLE`), and (iii) cross-level inconsistencies, such as an `ALLOW` on an entity whose screen is hidden. Errors block generation, while other issues are reported as warnings. At run time, remaining conflicts are resolved by the deny-overrides strategy, ensuring only consistent permissions over `NamedElements` reach enforcement layers.

3.2 Generation Architecture

The end-to-end generation pipeline is illustrated in Figure 4. The pipeline accepts three models as input: a Structural UML-like model that defines domain entities, an IFML-like GUI model that describes the user interface, and a PolicyDSL model that defines access rules on top of one, the other or both of those models. These models are processed by the data-driven app generator, which coordinates back-end, front-end, and policy artifact generation. When a policy model is provided, the generator produces a complete full-stack application consisting of a FastAPI-based back-end and a React-based front-end. Furthermore, intermediate policy representations are generated and shared across layers, depending on whether the policies are defined over the Structural model (back-end-oriented) or the GUI model (front-end-oriented).

As introduced earlier, the pipeline supports two enforcement scenarios. In Scenario A (back-end-first enforcement), access control policies are enforced at the back-end level using a FastAPI middleware that validates incoming requests and applies authorization decisions at run-time; the resulting decisions are then propagated to the front-end. In Scenario B (front-end only enforcement), policies are enforced within the front-end using a React-based policy context that dynamically adapts the user interface by hiding or disabling UI components, restricting access to screens, and masking sensitive data. Together, these scenarios illustrate the flexibility of the approach while preserving a unified policy definition.

To illustrate the approach, the running example introduced in Section 2 is used. The Structural (Figure 1), GUI (Figure 2), and PolicyDSL (Table 1) models define domain entities, user interface elements, and access control rules, respectively. Policies are specified as rules over subjects, resources, permissions, and effects, covering screen-, action-, and field-level constraints. Based on these models, the pipeline produces a unified application in which the same UI adapts dynamically to different roles at run-time. For instance, the *Librarian* role is granted full CRUD capabilities, the *Faculty* role experiences restricted actions and masked sensitive attributes (e.g., Release date), and the *Student* role is further restricted through limited screen access.

In the following, the initial phase of the policy transformation process (**Phase 1** in Figure 4) is described, followed by a detailed presentation of the two enforcement scenarios (**Phase 2** in Figure 4). It also introduces the model-to-UI traceability strategy.

3.2.1 Policy Transformation. The initial phase of the process involves the transformation of PolicyDSL models into structured JSON representations, as illustrated in Figure 4. The pipeline generates one of two specialized representations, aligned with the targeted enforcement scenario, depending on whether the policy is defined over the Structural model or the GUI model.

Under back-end-first enforcement (Scenario A), the transformation process yields a back-end-oriented policy JSON that encapsulates roles, targets (resources), permissions, and rules in a unified structure. Policy targets are directly derived from B-UML `NamedElements`, ensuring consistency with the underlying domain model and enabling evaluation at both entity and attribute level.

In the context of Scenario B (front-end-only enforcement), the transformation process yields a JSON oriented to UI that is specifically designed for client-side adaptation. In this representation, resources are automatically classified into different categories of UI components such as screens, buttons, and fields (data binding) based on their originating model elements. This eliminates the need for manual annotations and directly supports fine-grained UI control. The use of a JSON-based intermediate representation also facilitates interoperability with other low-code platforms by providing a platform-independent format. This opens the way to future work on reusing and migrating policies across different LCPs, such as from BESSER to other platforms.

3.2.2 Scenario A (back-end-first enforcement). The generated FastAPI back-end integrates a policy evaluation module injected through templates. Each request is associated with a role provided through an HTTP header, and access is validated before executing any CRUD operation. The evaluation function loads the policy JSON and applies deny-overrides semantics over matching rules. In the absence of applicable rules, the system defaults to a denial position, thus

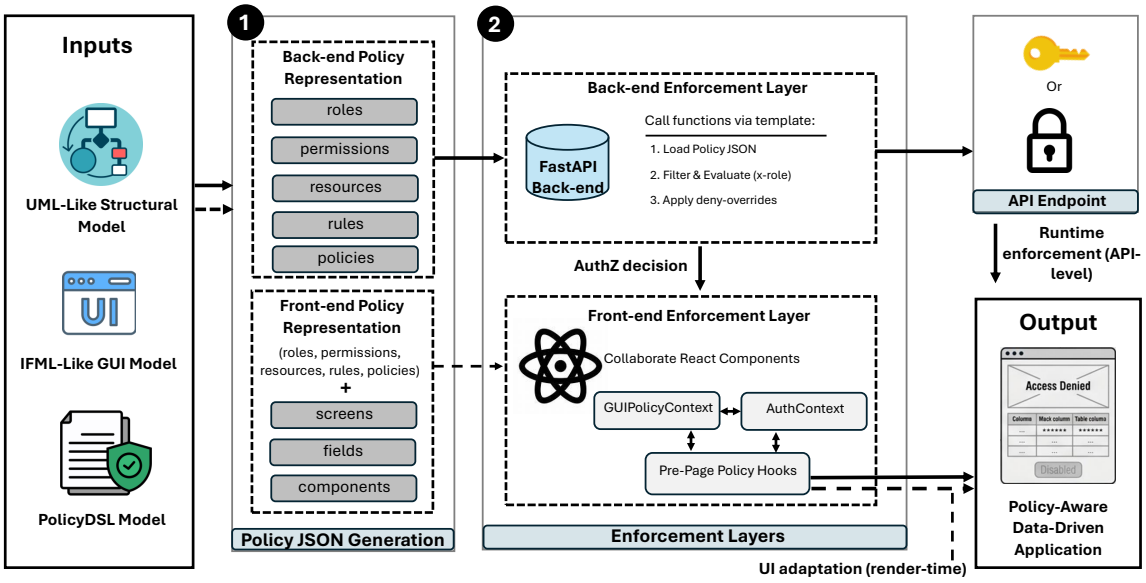


Figure 4: Policy-Aware Data-Driven Application Generation Pipeline (solid arrows: Scenario A; dashed arrows: Scenario B).

enforcing a closed-world assumption at the API level. A wrapper function converts the evaluation results into HTTP responses, ensuring consistent enforcement across all endpoints. Listings 2 shows the core back-end policy evaluation logic. The generated FastAPI module includes two functions injected via Jinja[3] templates. The *evaluate_policy()* function loads the policy, filters rules by role, resource, and permission, and enforces a deny-overrides strategy under a closed-world assumption (no match implies denial). The *require_policy()* function invokes this check and raises an HTTP 403 error if access is denied. Each generated endpoint calls this function before executing any operation, with the user role provided via an *x-role* HTTP header.

```python
def evaluate_policy(role, resource, permission):
    rules = load_policies().get("rules", [])
    relevant = [ r for r in rules
                 if r["subject"] == f"role_{role.lower()}"
                 and resource_matches(r["resource"], resource)
                 and r["permission"].upper() == permission.upper()]
    if any(r["effect"] == "DENY" for r in relevant): return False
    return any(r["effect"] == "ALLOW" for r in relevant)
def require_policy(user, resource, permission):
    if not evaluate_policy(user.role, resource, permission):
        raise HTTPException(status_code=403)
```

Listing 2: Generated back-end policy evaluation (simplified).

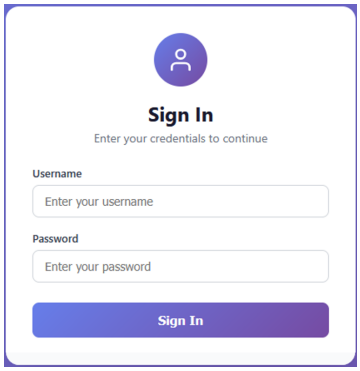
3.2.3 Scenario B (front-end-only enforcement). As illustrated in Figure 4, the front-end enforcement layer is generated as a set of React components that operate over the same policy specification. A central *GUIPolicyContext* loads the policy JSON once at the start of the application and exposes a set of evaluation functions that determine whether a resource should be visible, enabled, or masked. These functions implement the same deny-overrides semantics as the back-end but operate at render time in order to dynamically adapt the UI. An *AuthContext* provides role management. The active role is stored in *localStorage* and sent in an *x-role* header with each request, ensuring consistent role identity between front-end UI decisions and back-end enforcement. The generator also produces *page-level hooks* that pre-compute UI transformations, including column filtering, widget control, masking of sensitive fields, and action control (e.g., create/edit/delete permissions). Unauthorized screen access results in an access-denied view instead of the interface. In this scenario, enforcement is performed at the UI level, assuming a trusted client environment, and is intended for UI adaptation rather than as a security boundary.

```javascript
const filteredColumns = useMemo(() => {
  return allColumns
    .filter(col => {
      if (col.field === "email"
        && shouldHide("Author_email", userRole))
        return false;
      return true;})
    .map(col => {if (col.field === "email"
                  && shouldMask("Author_email", userRole))
                  return { ...col, mask: true };
                return col;
    });}, [allColumns, shouldHide, shouldMask, userRole]);
```

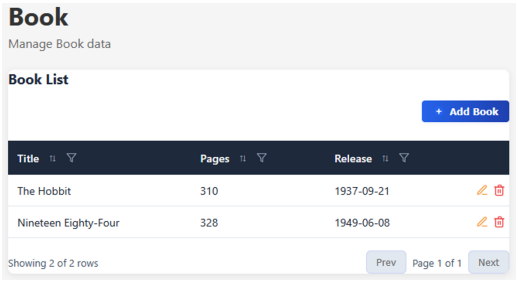
Listing 3: Generated React hook for policy-driven column filtering (simplified).

Listing 3 shows a simplified React hook for policy-driven column filtering in the *Author* page, where columns are dynamically removed or masked based on the active role and policies. In the current prototype, roles are passed via a client-side header for simplicity. The approach is compatible with OAuth 2.0/OIDC and SSO-based authentication, where roles are derived from server-validated token claims; integration with enterprise identity providers is left for future work. Figure 5 shows the generated UI pages for the *Library Management* example. The sign-in page is presented in Figure 5a. For the *Librarian* role (Figure 5b), all fields and CRUD actions are available. In contrast, the *Faculty* role (Figure 5c) has restricted editing and masked sensitive attributes (e.g., *Release Date*). Similarly, in Figure 5d, sensitive data such as the *Author* email is masked. These variations demonstrate how a single interface dynamically adapts based on access control policies.

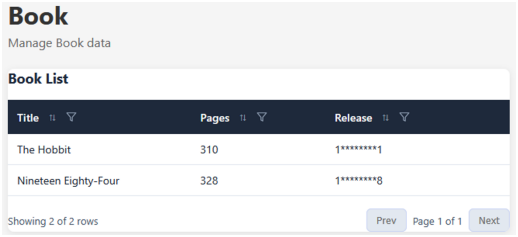
[3]https://jinja.palletsprojects.com/en/stable/



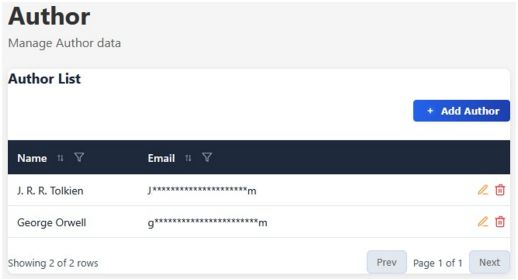
(a) Sign-in page.



(b) Librarian: full access (Book Page).



(c) Faculty: restricted actions (Book Page).



(d) Faculty: masked data (Author Page).

Figure 5: Role-based UI adaptation under policy enforcement.

3.2.4 Model-to-UI Traceability Strategy. To ensure traceability between PolicyDSL models and the generated UI, a lightweight mapping strategy is used during code generation. Since UI elements are derived from B-UML `NamedElements`, traceability is preserved at the model level. Generated UI components follow naming conventions derived from the domain and GUI models. This enables policy targets defined over `NamedElements` to be resolved consistently at generation time. For instance, action-based UI elements are linked to operations through their semantic names, while data fields inherit the names of the corresponding structural properties. This enables automatic binding between policies and UI elements without manual configuration. However, it depends on consistent naming in generated artifacts, and more robust identifier-based mappings are left for future work.

4 Tool Support

The approach is implemented as an extension of the BESSER low-code platform, following a modular, model-driven architecture. The toolchain integrates Structural (UML-like), GUI (IFML-like), and PolicyDSL models to automatically generate secure, policy-aware data-driven applications. It follows a pipeline of loosely coupled components. At design time, models are processed by a model processor and policy interpreter, which feed back-end and front-end generators to produce secured APIs and adaptive user interfaces. At run-time, policies are externalized (e.g., JSON) and interpreted dynamically, enabling incremental updates. When a policy changes, only affected components are updated, and the application adapts automatically through reloading mechanisms. This reduces the effort to maintain and evolve policies and supports rapid experimentation without full redeployment. The modular design also facilitates extensions with new policy types and enforcement strategies. The full pipeline is publicly available[4].

5 Evaluation

This section presents a fully automated and reproducible evaluation of the proposed approach under two enforcement scenarios: **Scenario A (back-end enforcement)**, where policies are enforced through FastAPI middleware and propagated to the front-end, and **Scenario B (front-end enforcement)**, where policies are enforced directly in the UI using a React-based Policy context. The evaluation addresses four research questions:

RQ1 (Correctness): Are the generated artifacts structurally consistent with the policy model and do they correctly enforce its intended behavior?
RQ2 (Security Robustness): Are the generated artifacts robust against policy bypass attempts and malformed or adversarial inputs at run-time?
RQ3 (Run-time Performance): What is the run-time overhead of policy enforcement across back-end and front-end layers?
RQ4 (NL Transformation Correctness): To what extent can natural-language access-control descriptions be correctly translated into PolicyDSL rules?

Datasets and Experimental Setup. The evaluation is based on a *Library Management* case study with three entities (*Book, Author, Library*). We generate 200 policy variants, evenly distributed across the two enforcement scenarios. All variants share the same Structural and GUI models, while the policy model is varied. Variants differ in number of roles, rules, and resources, the ratio of `ALLOW/DENY` decisions, and policy granularity (coarse: screen-level;

[4] **GitHub Repository:** https://github.com/BESSER-PEARL/BESSER-PolicyDSL.

Table 2: Benchmark configuration and aggregated statistics across policy granularity levels.

Granularity	#Instances	Avg. Roles	Avg. Rules	Deny Ratio	Avg. Resources	Avg. Perm.	Avg. Concept Count	Avg. FE* Points
Coarse	68	7	52	0.25	7	3	19	14
Medium	66	6	89	0.35	13	4	25	20
Fine	66	6	104	0.38	16	4	28	27
Total	200	6.3	81.6	0.32	12	3.6	24	20.33

* Front-end Enforcement Points

Table 3: Behavioral correctness (decision-level oracle).

Layer	Decisions	Matched	Accuracy
Back-end	9921	9921	100%
Front-end	6921	6921	100%
Adversarial	300	300	100%
Total	17142	17142	100%

medium: screens and actions; fine: screens, actions, and fields/UI widgets). A seed-controlled generation process is used to ensure diversity and reproducibility. The deny ratio controls policy restrictiveness by balancing allowed and denied interactions. All three research questions are evaluated over the same 200 variants and their corresponding generated application instances. To ensure reproducibility, all artifacts—including models, generated applications, results, and evaluation scripts—are publicly available[5].

The policy complexity is defined as follows:

$$\text{concept_count} = |\text{roles}|+|\text{resources}|+|\text{permissions}|+|\text{effects}| \quad (1)$$

A summary of the benchmark configuration is reported in Table 2, showing the parameter ranges used to generate the 200 policy variants. All experiments were executed on a Windows 11 machine with an Intel i7-1280P CPU and 32 GB RAM.

5.1 RQ1 (Correctness)

Correctness is evaluated at the level of access-control semantics, complemented by a structural validation of generated artifacts. The structural phase verifies that PolicyDSL rules are correctly translated into executable artifacts and that enforcement hooks are properly integrated in both scenarios, achieving consistent 100% conformance across all generated instances. The main focus of RQ1 is behavioural correctness, i.e., whether run-time enforcement produces the intended access-control decisions.

RQ1 is evaluated over the same 200 variants used in RQ2–RQ3. For each variant, we enumerate its induced decision space and verify that the generated enforcement produces the expected outcome for every target. The space includes all (role, resource, permission, effect) combinations derived from the policy. Across the 200 variants, these decision spaces vary in size depending on the number of roles, rules, and resources, and span the full range of policy granularities considered in the study (coarse, medium, and fine).

A decision-level oracle independently validates enforcement semantics. For each variant, a ground-truth set of expected outcomes is derived directly from the policy specification, representing the intended `ALLOW/DENY` or UI-level effects. All expected decisions are replayed and compared against oracle outputs, producing a full enumeration of the decision space. For instance, the PolicyDSL rule stating that *Faculty is denied access to BookEditButton* is reflected in the evaluation tuple *(Faculty, BookEditButton, button)*, where both expected and actual outcomes are *FALSE*, confirming correct enforcement at the UI level.

In addition, we test three adversarial cases per variant: conflicting rules (`ALLOW+DENY` $\rightarrow$ `DENY`), unknown roles (default-deny), and incomplete bindings. All 300 checks pass, confirming correct handling of edge cases. Overall, across the 200 variants, this yields 9,921 + 6,921 decisions and 300 adversarial checks (17,142 total), all matching the expected outcomes (100% accuracy; Table 3).

5.2 RQ2 (Security Robustness)

To complement functional correctness, the evaluation assesses whether generated enforcement artifacts *fail securely* under adversarial conditions. A fully automated security testing harness is executed over all 200 generated application variants, resulting in 3,200 validation checks across front-end and back-end run-time behaviors. For each variant, a deterministic test generation pipeline derives a set of adversarial and structural test cases across six security dimensions. These tests are executed against running application instances or their corresponding policy evaluation layers, covering both back-end request enforcement and front-end rendering behavior logic. No manual interaction is required. All test cases are derived programmatically from the original policy specification, ensuring full traceability between policy structure and executed checks. The evaluation consists of six dimensions:

Secure Defaults. Adversarial run-time conditions are injected during application initialization, including undefined roles, uninitialized policy states, and missing resource identifiers. Each case is executed across layers to verify deny-by-default semantics, including correct UI rendering suppression and enforcement of default-deny decisions prior to policy availability.

Code Security. A static analysis pass is combined with run-time verification over the generated TypeScript/React application. Runtime verification ensures that authorization decisions are executed

[5]**GitHub Repository:** https://github.com/BESSER-PEARL/BESSER-PolicyDSL.

Table 4: Security robustness evaluation across 200 variants (3,200 checks total).

Dimension	Variants	Checks	Passed	Rate
Secure Defaults	200	600	600	100%
Code Security	200	700	700	100%
JSON Robustness	200	1000	1000	100%
Role Isolation (front-end)	100	300	271	90.3%
Deny-Override (back-end)	100	300	300	100%
Role Isolation (back-end)	100	300	296	98.7%
Total	200	3200	3167	99.0%

through the centralized policy context and not bypassed by direct policy access.

JSON Robustness. Policy inputs are mutated into edge cases (empty objects, missing `rules`, null fields, and adversarial extensions). Each case is executed through the evaluation layer and must terminate without error under default-deny semantics.

Role Isolation. Role-based access is reconstructed from policy rules as (role, resource, permission) triples. Differences between induced access sets are evaluated to verify least privilege and inter-role separation.

Deny-Override (back-end). Conflicting rules with identical (subject, resource, permission) keys are grouped and evaluated using a back-end decision oracle implementing deny-overrides-allow with default deny.

All checks are executed automatically on generated artifacts and policy evaluation logic. Table 4 shows 100% success for secure defaults, code security, JSON robustness, and deny-override. Minor deviations appear in role isolation due to conservative heuristics, with no actual bypasses observed.

5.3 RQ3 (Run-time Performance)

The run-time overhead introduced by policy enforcement during application execution is evaluated across 200 generated application variants. Each variant is executed in two configurations: a baseline version without policy checks and a policy-aware version where requests are intercepted by the evaluation function. The two configurations differ only in the presence of the policy layer, allowing the isolation of enforcement overhead. Each request is executed 40 times, and for Scenario A (back-end), median latency is reported. For Scenario B (front-end), policies are enforced directly in the user interface. The in-memory latency of evaluating policy conditions during rendering and interaction is measured. Across both scenarios, the evaluation is conducted over 200 variants, where the number and structure of policy rules are systematically varied.

For Scenario A, the median request latency increases from 19.785 ms (baseline) to 19.9405 ms (policy-aware), corresponding to an absolute overhead of 0.1555 ms (0.79%). For Scenario B, policy evaluation remains lightweight, with latencies ranging from 3.3μs to 26.6μs across variants. Figure 6 illustrates the relationship between the number of rules and the latency of the policy-layer. Overall, policy enforcement introduces a small and bounded overhead across all 200 variants for both scenarios, preserving responsiveness while enabling fine-grained multi-layer access control.

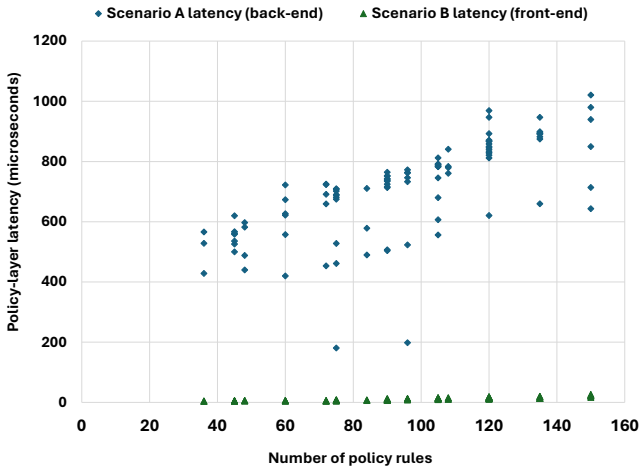


Figure 6: Policy-layer latency vs number of rules.

Table 5: Evaluation results for NL policy authoring.

Extension	Instances	Precision	Recall	Accuracy
NL (Back-end)	36 rules	1.00	1.00	1.00
NL (Front-end)	39 rules	1.00	1.00	1.00

5.4 RQ4 (NL Transformation Correctness)

PolicyDSL rules can be authored programmatically or in natural-language (Section 3.1.2). RQ4 evaluates the correctness of the NL path, i.e., whether the LLM produces the intended PolicyDSL rules. This is a focused study on the *Library Management* running example (not the 200 variants), as it requires a manually defined ground truth. For each scenario, we define NL descriptions and derive the corresponding gold rules. Generated rules are compared to the gold standard using their (subject, resource, permission, effect) tuples. A rule is correct only if all four elements match; we also track missing, wrong-effect, and spurious rules, and report precision and recall. The evaluation includes 36 back-end and 39 front-end rules (75 total). All rules are correctly generated, with no missing or incorrect cases, yielding precision = recall = F1 = 1.00 in both scenarios (Table 5). These results confirm correct transformation within this constrained setting. The evaluation covers 75 rules in a single domain, and given known variability in LLM outputs, it does not support general claims of reliability. Broader evaluation is left for future work.

6 Related Work

This section provides a review of related studies pertinent to our research, grouped into different categories.

Model-driven Security and RBAC Integration. Model-driven security approaches such as SecureUML [13] established the foundations for integrating RBAC specifications into UML-based development. These approaches model authorization constraints alongside structural models, enabling automated transformation into enforcement infrastructures. However, they remain largely backend-oriented, with limited support for modern front-ends and no explicit

Table 6: Comparison of PolicyDSL with related approaches.

Approach	Feature-Level	UI Widget-Level	Front-end Support	Cross-Layer	Automation	Model-Driven
SecureUML [13]	✓	✗	✗	✗	✓	✓
Basin et al. [4]	✓	△	✗	✗	✓	✓
IFML / WebML	△	✗	✓	✗	✓	✓
XACML	✓	✗	✗	△	✗	✗
Low-code platforms	✓	△	✓	△	△	△
Dandelion+ [15]	✓	△	△	✗	✓	✓
PolicyDSL (this work)	✓	✓	✓	✓	✓	✓

Abbreviations: ✓ Full support; ✗ No support; △ Partial support.

treatment of fine-grained GUI behaviours such as widget visibility, masking, or interaction control. UMLsec [11] extends UML with security stereotypes to capture properties such as confidentiality and integrity but focuses on system-level requirements rather than executable access control for user interfaces, and does not provide mechanisms for generating enforcement logic, particularly at the front-end level. IFML [17] and WebML provide rich abstractions for user interaction, navigation, and data binding in data-driven applications; however, access control is not treated as a first-class concern and is typically delegated to external mechanisms or manual implementation. Extensions of WebML with RBAC focus on resource-level authorization tied to domain entities or interaction patterns, without providing a unified GUI-aware policy model.

Multi-layer Access Control and Policy Languages. Basin et al. [4] propose the integration of security concerns into GUI generation by propagating back-end permissions to the interface level. In such approaches, the GUI is used to display the results of the back-end authorization decisions. While this improves consistency, the model remains data-centric, which limits expressiveness for UI-specific behaviours such as conditional rendering, masking, or interaction-level constraints. XACML[6] provides a standard for ABAC and has been studied in policy-based and model-driven systems, including prior MDE integrations. It supports complex policies with the combination of algorithms and contextual conditions. However, it focuses primarily on policy decision-making at the enforcement point and does not explicitly capture UI elements or UI-level adaptation, which limits its direct use in model-driven pipelines requiring fine-grained front-end control.

Access control in Low-Code platforms. Recent studies [15, 16] on low-code development platforms highlight the importance of access control as a fundamental platform capability. Platforms such as OutSystems, Mendix, and Power Apps provide role-based authorization mechanisms across application, data, and screen levels. However, access control is typically embedded within platform-specific configurations and lacks a unified, model-driven representation. For example, in Mendix, policies are implicitly encoded through combinations of entity rules, page visibility, and role configurations, rather than through a unified policy model. While feature masking and UI restrictions can be achieved, they are often coarse-grained and lack a declarative and systematic representation. As a result, coordination between front-end and back-end enforcement remains limited, and traceability of policies is reduced. Model-driven approaches to low-code platform engineering, such as Dandelion+ [15], emphasize the need for explicit modeling of roles and permissions, as traditional domain models fail to adequately capture access control concerns. However, these approaches focus primarily on platform architecture and workflows, with limited support for the unified specification and enforcement of fine-grained access control policies across multiple application layers.

Table 6 compares the reviewed approaches with our framework across several key dimensions, including *feature-level* support (the ability to define access control policies at the back-end or resource level), *UI widget-level* support (fine-grained control over front-end elements such as visibility and interaction), *front-end* support (modeling and/or generation of the user interface layer), *cross-layer enforcement* (the coordinated enforcement of policies across both front-end and back-end layers), as well as *automation* capabilities (the capability of automatically transforming policy specifications (models) into executable artifacts), and *model-driven* principles.

As demonstrated in the table, existing approaches typically provide support for either back-end policy specification (e.g., SecureUML, IFML/WebML) or general-purpose enforcement mechanisms (e.g., XACML) but lack comprehensive support for fine-grained UI-level control and integrated front-end enforcement. Although some approaches, such as low-code platforms and Dandelion+, partially address user interface concerns, they do not offer a unified, model-driven representation that systematically links policy specification to code generation. In contrast, our approach, PolicyDSL, provides full support across all considered dimensions by enabling explicit modeling of access control policies at both feature and widget levels, and by ensuring consistent enforcement across layers through a model-driven and automated approach. This support helps PolicyDSL connect policy definition and enforcement by making access control part of the modeling process. By combining back-end and front-end constraints in one metamodel, it improves consistency and reduces security risks from fragmented implementations. The approach is practical and extensible, and since the metamodel and JSON representation are platform-independent, reuse only requires adapting the generation templates.

7 Extension of the Approach

To demonstrate extensibility, we introduce AI-agent-aware access control, treating agents as first-class subjects. The extension reuses the core metamodel and adds agents, agent policies, and delegation rules linking users to agents for specific resources and permissions. A delegated action is allowed only if the user is permitted, the agent policy allows it, and a matching delegation exists. Delegations may

[6] https://en.wikipedia.org/wiki/XACML

include constraints (e.g., approval or no autonomous execution) and cannot exceed the delegator's permissions, preventing privilege escalation. These checks build on the existing deny-overrides mechanism. This is a proof of concept. A preliminary check on the running example (valid delegation, missing approval, conflicting policies, invalid agents) behaves as expected; full specification and evaluation are left for future work.

8 Limitations and Threats to Validity

Limitations. The current implementation mainly supports role-based policies. The metamodel is extensible toward ABAC, but attribute-based subjects and contextual conditions (e.g., time, location) are not yet fully supported. Claims on correct enforcement, robustness, and low overhead are based on a single case study and internal comparisons. Broader evaluations, including comparisons with manual implementations, generalization to other domains, and more real-world case studies, are left for future work.

Threats to Validity. The evaluation may be influenced by the representativeness of the data set; to mitigate this, we constructed a benchmark that covers various access control policies across multiple granularity levels, roles, and configurations. Internal validity may be affected by implementation or experimental bias; this is reduced by executing all policy variants through a unified, fully automated pipeline, ensuring consistent and comparable results. To support reproducibility, all evaluation artifacts and results are publicly available[7].

9 Conclusion and Future Work

This paper presents a model-driven, policy-aware approach for fine-grained access control across data-driven applications. PolicyDSL enables unified specification over data and UI elements with consistent back-end and front-end enforcement in the BESSER framework. Evaluation over 200 policy variants spanning coarse-, medium-, and fine-grained levels confirms correct multi-layer enforcement, robustness against adversarial inputs, and low run-time overhead, and a focused study shows that the natural-language authoring path reproduces the intended PolicyDSL rules. The evaluation is conducted on one case study and should be interpreted as a proof of concept; validation on independent real-world systems is part of future work.

Future work will extend the approach with richer policy models, including attribute-based and context-aware constraints (e.g., time and location), and integrate formal methods for policy validation, conflict detection, and user-centered workflows. We also plan to evaluate the approach on independent, real-world case studies, conduct usability studies, and assess scalability and industrial adoption, as well as to further develop the AI-agent-aware access control extension. In addition, we will develop a graphical editor for defining PolicyDSL specifications via a concrete visual syntax.

Acknowledgments

This research is supported by the Luxembourg National Research Fund (FNR) through the PEARL program under grant agreement 16544475.

[7] **GitHub Repository:** https://github.com/BESSER-PEARL/BESSER-PolicyDSL.

References

[1] Sara Aboukadri, Aafaf Ouaddah, Abdellatif Mezrioui, and Ikram El Asri. 2025. Leveraging RAG and LLMs for Access Control Policy Extraction From User Stories in Agile Software Development. *IEEE Access* (2025).
[2] Matthew Oladeji Ajimati, Noel Carroll, and Mary Maher. 2025. Adoption of low-code and no-code development: A systematic literature review and future research agenda. *Journal of Systems and Software* 222 (2025), 112300.
[3] Iván Alfonso, Aaron Conrardy, Armen Sulejmani, Atefeh Nirumand, Fitash Ul Haq, Marcos Gomez-Vazquez, Jean-Sébastien Sottet, and Jordi Cabot. 2024. Building BESSER: an open-source low-code platform. In *International Conference on Business Process Modeling, Development and Support*. Springer, 203–212.
[4] David Basin, Manuel Clavel, Marina Egea, Miguel A García de Dios, Carolina Dania, Gonzalo Ortiz, and Javier Valdazo. 2011. Model-driven development of security-aware GUIs for data-centric applications. In *International School on Foundations of Security Analysis and Design*. Springer, 101–124.
[5] Mairon Belchior, Daniel Schwabe, and Fernando Silva Parreiras. 2012. Role-based access control for model-driven web applications. In *International Conference on Web Engineering*. Springer, 106–120.
[6] Alessandro Bozzon, Tereza Iofciu, Wolfgang Nejdl, Antonio Vincenzo Taddeo, and Sascha Tönnies. 2007. Role Based Access Control for the Interaction with Search Engines.. In *COOPER*.
[7] Jordi Cabot. 2020. Positioning of the low-code movement within the field of model-driven engineering. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 1–3.
[8] Nastaran Farhadighalati, Luis A Estrada-Jimenez, Sanaz Nikghadam-Hojjati, and Jose Barata. 2025. A systematic review of access control models: Background, existing research, and challenges. *IEEE Access* 13 (2025), 17777–17806.
[9] Juliana George. 2024. Implementing Role-Based Access Control (RBAC) in Java and AngularJS Web Applications. (2024).
[10] Vincent C Hu, David Ferraiolo, Rick Kuhn, Adam Schnitzer, Kenneth Sandlin, and Karen Scarfone. 2019. Guide to attribute based access control (abac) definition and considerations. *NIST special publication* 800, 162 (2019), 1–47.
[11] Jan Jürjens. 2005. *Secure systems development with UML*. Springer.
[12] Nadine Kashmar, Mehdi Adda, Mirna Atieh, and Hussein Ibrahim. 2021. Access control metamodel for policy specification and enforcement: From conception to formalization. *Procedia Computer Science* 184 (2021), 887–892.
[13] Torsten Lodderstedt, David Basin, and Jürgen Doser. 2002. SecureUML: A UML-based modeling language for model-driven security. In *International Conference on the Unified Modeling Language*. Springer, 426–441.
[14] Khang Mai, Nakul Ghate, Jongmin Lee, and Razvan Beuran. 2025. LLM-Based Fine-Grained ABAC Policy Generation.. In *ICISSP (2)*. 204–212.
[15] Francisco Martínez-Lasaca, Pablo Díez, Esther Guerra, and Juan de Lara. 2026. A model and workflow-driven approach for engineering domain-specific low-code platforms and applications: F. Martínez-Lasaca et al. *Software and Systems Modeling* 25, 2 (2026), 579–614.
[16] Jean-Marie Mottu and Gerson Sunyé. 2024. Emerging New Roles for Low-Code Software Development Platforms. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*. 905–914.
[17] OMG. 2025. O.M.G.: About interaction flow modeling language specification. Retrieved June 11, 2024 from https://www.omg.org/spec/IFML
[18] Elena Planas, Salvador Martínez, Marco Brambilla, and Jordi Cabot. 2023. Modeling and enforcing access control policies in conversational user interfaces. *Software and Systems Modeling* 22, 6 (2023), 1925–1944.
[19] Ravi S Sandhu. 1998. Role-based access control. In *Advances in computers*. Vol. 46. Elsevier, 237–286.
[20] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. 2002. Role-based access control models. *Computer* 29, 2 (2002), 38–47.
[21] Shang Siyuan, Liu Aodi, Du Xuehui, Wang Xiaohan, and Tan Ming. 2025. ABAC policy mining method for heterogeneous access control system: S. Siyuan et al. *The Journal of Supercomputing* 81, 9 (2025), 1065.